A Constraint-Based Approach to Automatic Data Partitioning

Wonchan Lee
Computer Science Department
Stanford University

Advanced Modeling & Simulation (AMS) Seminar Series
NASA Ames Research Center, September 5th, 2019

Auto-Parallelizers for Distributed Systems

- Goal: automatically generate distributed memory code from sequential programs
- Focus on data parallel programs
- Numerous efforts in the past several decades
    - High Performance Fortran and its predecessors (Fortran D, Vienna Fortran)
    - Polyhedral compilers for distributed memory machines

Why are they not being used more widely?

Issue 1: Configurability

- Compilers have to make decisions without the information only available at runtime

```
for i in A:
  A[i] = f(i)

for i in B:
  B[i] = g(A[i+1])
```

Sequential Code

```
parallel for p = 0, P:
  int S = N/P              // N = |A|
  int lA[S+1], lB[S]

  for i = 0, S:
    lA[i] = f(p*S+i)

  if p > 1:   send(p-1,  lA[0])
  if p < P-1: recv(p+1, &lA[S])

  for i = 0, S:
    lB[i] = g(lA[i+1])
```

Partitioning of **A** and **B**

Decomposition of loop iterations

Inter-node communication

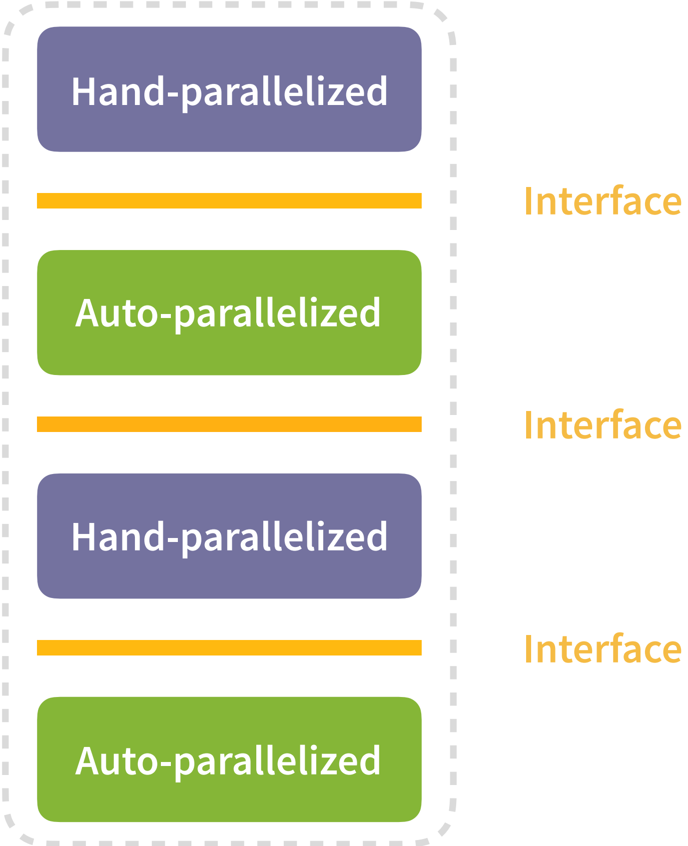
Distributed Code

- Decomposition of program and data must be determined at compile time, often by **hard-coded heuristics** in the compiler

- **Indirect accesses** make the output program even more obscure (inspect/executor)

Issue 2: Composability

In practice, programs look like this:



We need **interface** for seamless integration,
but auto-parallelized parts are **opaque** to the rest of the program

Data Distributions in HPF

- Annotation language to describe the primary partition of data
  - E.g., tiling on the first dimension of A:

```
      REAL A(1000,10000)
!HPF$ DISTRIBUTE A(BLOCK,*)
```

- Can serve as an interface for both configuration and composition
  - Support for sharing data partitions is key to configurability and composability
- Limited because “data distributions were not themselves data objects”†

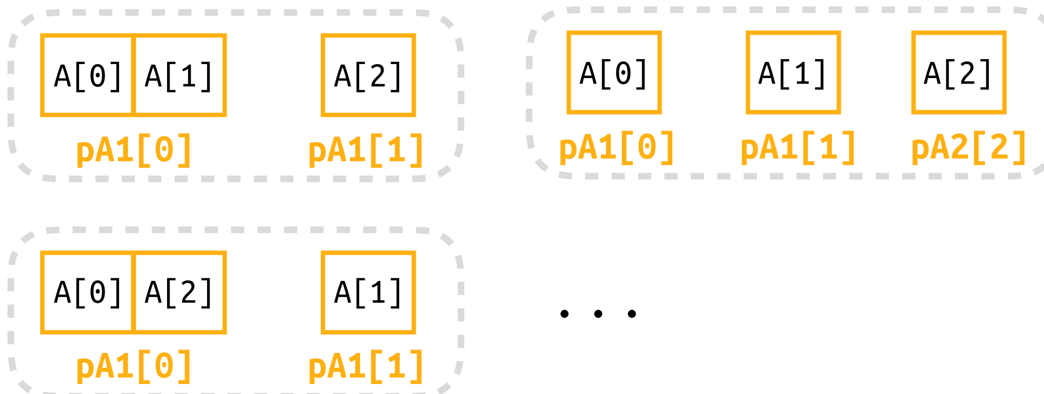
† Ken Kennedy, Charles Koelbel, and Hans Zima. 2007. The Rise and Fall of High Performance Fortran: An Historical Object Lesson. In Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages. ACM, 7–1.

Programming Models with First-Class Partitions

Use data partitions as programmable objects

```
// pA1 is a partition of A
parallel for x in pA1:
  A = pA1[x]
  for i in A:
    A[i] = f(i)
```

pA1 is an abstraction over partitions of A constructed at runtime:



Examples: **Legion**, StarPU, PaRSEC

Data partitions can naturally serve as an interface between different parts

Programming Models with First-Class Partitions

Can provide synchronization and communication from **multiple data partitions**

```
// pA1 and pA2 are partitions of A
parallel for x in pA1:
  A = pA1[x]
  for i in A:
    A[i] = f(i)

parallel for x in pB:
  A = pA2[x]
  B = pB[x]
  for i in B:
    B[i] = g(A[i+1])
```

Implicit communication and synchronization for every `i` and `j` such that `pA1[i]` ∩ `pA2[j]` ≠ ∅

→ **Can be handled automatically** by compiler‡ or runtime system†

† Michael Bauer, Sean Treichler, Elliott Slaughter, Alex Aiken, *Legion: expressing locality and independence with logical regions*. SC12.

‡ E. Slaughter, W. Lee, S. Treichler, W. Zhang, M. Bauer, G. Shipman, P. McCormick, and A. Aiken, *Control replication: Compiling implicit parallelism to efficient SPMD with logical regions*, SC17.

Auto-Parallelization as Constraint Solving

Auto-parallelization amounts to finding **legal partitions** by solving **partitioning constraints**

```
parallel for x in pA1:
  A = pA1[x]
  for i in A: ①
    A[i] = f(i)

parallel for x in pB:
  A = pA2[x]
  B = pB[x]
  for i in B: ②
    B[i] = g(A[i+1]) ③
```

Find partitions **pA1**, **pA2**, and **pB** that satisfy these **constraints**:

1. pA1 covers A
2. pB covers B
3. For any index `i` in `pB[x]`, `pA2[x]` includes `i+1`

Constraint-Based Automatic Data Partitioning

Parallelizes sequential program using **data partitions**

Infers **partitioning constraints**

Discharges constraints with **interface constraints**

```
...
// Hand-parallelized code
...
assert(π(some_pA))
```

```
for i in A:
  A[i] = f(i)
```

```
require(π(pA))
parallel for x in pA:
  A = pA[x]
  for i in A:
    A[i] = f(i)
```

Or, **synthesizes partitioning code** using constraint solver

```
pA = some_pA
parallel for x in pA:
  ...
```

```
pA = partition(A,...)
parallel for x in pA:
  ...
```

DPL as Constraint Language

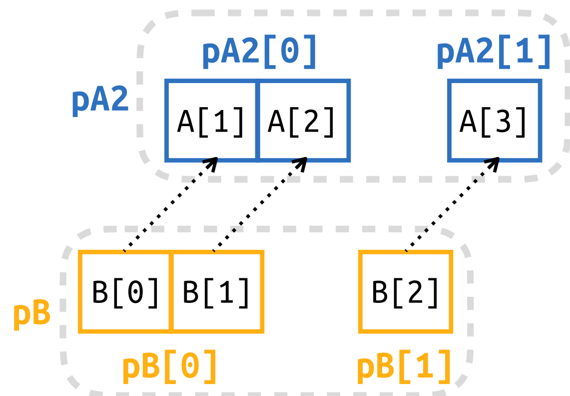
- DPL(Dependent Partitioning Language)[†]: domain specific language for data partitioning
  - DPL programs construct data partitions using high-level operators
  - DPL operators have well-defined semantics and scalable implementation
- DPL can be used to describe both partitioning constraints and their solutions

† Sean Treichler, Michael Bauer, Rahul Sharma, Elliott Slaughter, and Alex Aiken, *Dependent Partitioning,* OOPSLA16.

Constructing Partitions with DPL

Example:

```
parallel for x in pB:
  A = pA2[x]
  B = pB[x]
  for i in B:          ③
    B[i] = g(A[i+1])
```



Partition of the range of `λi.i+1`

Collecting image of `λi.i+1`

Partition of the domain of `λi.i+1`

③ $\forall$`j`, $\forall$`i` $\in$ `pB[j]`, `i+1` $\in$ `pA2[j]`

A function that maps `i` to `i+1`

DPL program: `pA2 = image(pB, λi.i+1)`

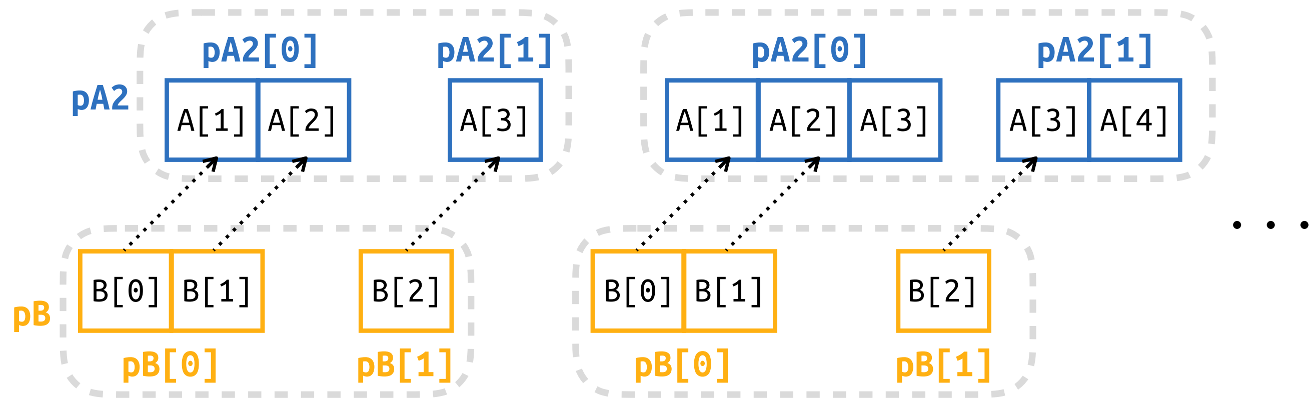
Characterizing Legal Partitions with DPL

Example:

```
parallel for x in pB:
  A = pA2[x]
  B = pB[x]
  for i in B:
    B[i] = g(A[i+1])
```

$\forall$j, $\forall$i $\in$ pB[j], i+1 $\in$ pA2[j]

Many partitions can satisfy the constraint



Constraint that characterize **all legal partitions** for **pA2**:

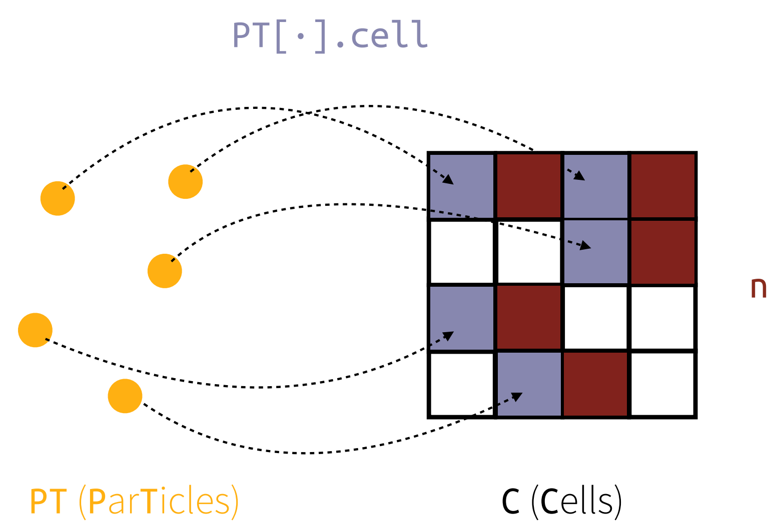
image(pB, λi.i+1) ⊆ **pA2**

The program **pA2 = image(pB, λi.i+1)** is one solution of this constraint

Example: Particle Simulation

Updates the position of every particle using the velocity of cells

```
for i in PT:
  c = PT[i].cell
  PT[i].pos = f(C[c].vel,C[n(c)].vel)
```



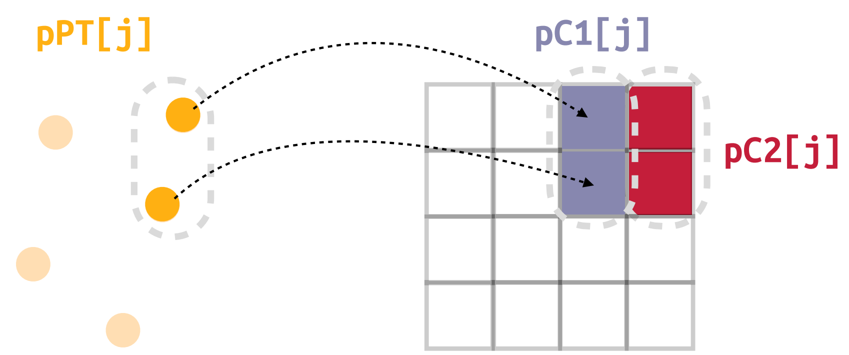
Constraint Inference

Identifies necessary data partitions

Needs two partitions **pC1** and **pC2** of C for different access patterns

Needs a partition **pPT** of PT

```
for i in PT:
  c = PT[i].cell
  PT[i].pos = f(C[c].vel,C[n(c)].vel)
```



$\forall \mathtt{j}, \forall \mathtt{i} \in \mathtt{pPT[j]},\ \mathtt{PT[i].cell} \in \mathtt{pC1[j]}$

$\forall \mathtt{j}, \forall \mathtt{i} \in \mathtt{pC1[j]},\ \mathtt{n(i)} \in \mathtt{pC2[j]}$

Infers partitioning constraints

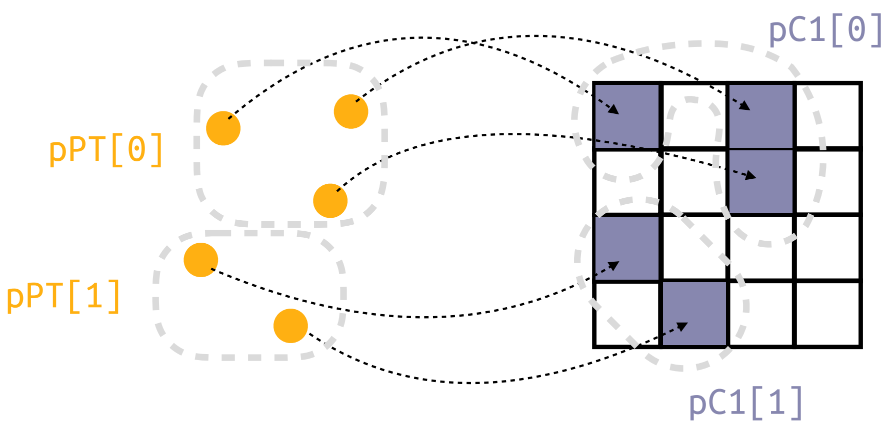
```
require(complete(pPT,PT))
require(image(pPT,PT[·].cell) ⊆ pC1)
require(image(pC1,n) ⊆ pC2)
parallel for x in pPT:
  PT = pPT[x]; C1 = pC1[x]; C2 = pC2[x]
  for i in PT:
    c = PT[i].cell
    PT[i].pos = f(C1[c].vel,C2[n(c)].vel)
```

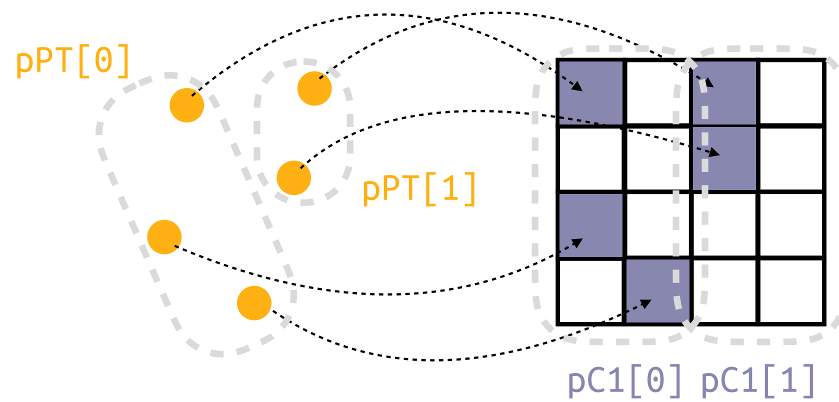
Must run all iterations at least once

A function that maps particles to cells

Solving Constraints

Two solutions for `require(image(pPT,PT[·].cell) ⊆ pC1)` :





✔
```
pPT = equal(PT,N)
pC1 = image(pPT,PT[·].cell)
```

```
pC1 = equal(C,N)
pPT = preimage(PT[·].cell,pC1)
```

```
require(complete(pPT,PT))
require(image(pPT,PT[·].cell) ⊆ pC1)
require(image(pC1,n) ⊆ pC2)
```

Solve constraints →

```
pPT = equal(PT,N)
pC1 = image(pPT,PT[·].cell)
pC2 = image(pC1,n)
```

Handling Multiple Loops

One loop = One set of partitioning constraints

```
for i in PT:
  c = PT[i].cell
  PT[i].pos = g(C[c].vel,C[n(c)].vel)
```

```
require(complete(pPT,PT))
require(image(pPT,PT[·].cell) ⊆ pC1)
require(image(pC1,n) ⊆ pC2)
```

```
for i in C:
  C[i].vel  = h(C[i].acc,C[n(i)].acc)
```

```
require(complete(pC3,C))
require(image(pC3,n) ⊆ pC4)
```

✔ Capture **all possible partitioning strategies**

✘ Can lead to **excessive communication** if solved naïvely

Handling Multiple Loops

Constraint solver **unifies** partitions to maximize partition reuse

```
for i in PT:
  c = PT[i].cell
  PT[i].pos = g(C[c].vel,C[n(c)].vel)
```

```
require(complete(pPT,PT))
require(image(pPT,PT[·].cell) ⊆ pC1)
require(image(pC1,n) ⊆ pC2)
```

```
for i in C:
  C[i].vel  = h(C[i].acc,C[n(i)].acc)
```

```
require(complete(pC3,C))
require(image(pC3,n) ⊆ pC4)
```

Similar access patterns

Isomorphic constraints

Unified!

Solution:

```
pC3 = pC1 = equal(PT,N)
pC4 = pC2 = image(pC1,n)
pPT = preimage(PT[·].cell,pC1)
```

External Constraints

In the real simulation code, particles might **move to different cells**,

```
while t < T:
  for i in PT:
    c = PT[i].cell
    PT[i].pos = g(C[c].vel,
                  C[n(c)].vel)

  for i in C:
    C[i].vel = g(C[i].acc,
                 C[n(i)].acc)

  for i in PT:
    PT[i].cell = h(PT[i].pos)
```

Solve constraints

requiring **pPT** to be **repartitioned** every time step ☹️

```
pC1 = equal(C,N)
pC2 = image(pC1,n)

while t < T:
  pPT = preimage(PT[·].cell,pC1)

  parallel for x in pPT:
    ...

  parallel for x in pC1:
    ...

  parallel for x in pPT:
    ...
```

External Constraints

User can **manually parallelize** particle transfer code
and provide external constraints as an interface:

```
while t < T:
  ...
  // Manual particle transfer code using
  // pParticle and pCell
  ...
  assert(
    image(pParticle,PT[·].cell) ⊆ pCell))
  ...
```

Solve constraints with
external constraints

```
pC3 = pC1 = pCell
pC4 = pC2 = image(pCell,n)

while t < T:
  ...
  // Manual particle transfer code
  ...
  pPT = pParticle
  ...
```

Partitioning constraints:

Unifiable constraints

No more repartitioning 🙂

```
require(complete(pPT,PT))
require(image(pPT,PT[·].cell) ⊆ pC1)
require(image(pC1,n) ⊆ pC2)
require(complete(pC3,C))
require(image(pC3,n) ⊆ pC4)
```

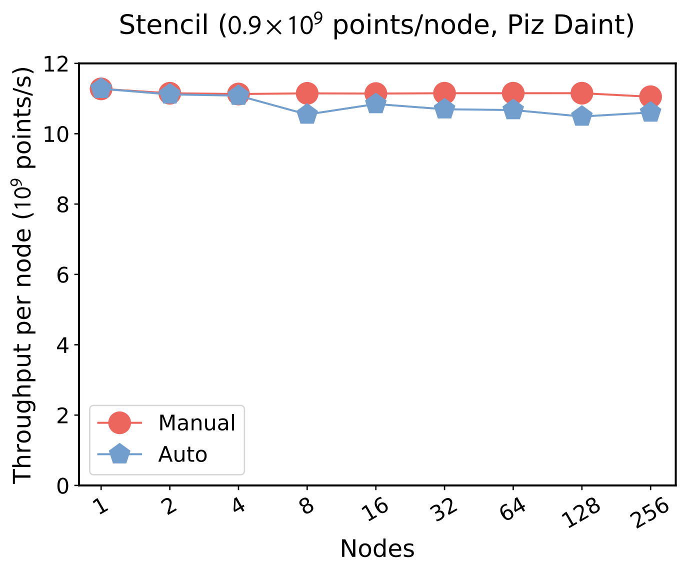
External constraints provide a precise control
over the automated data partitioning process

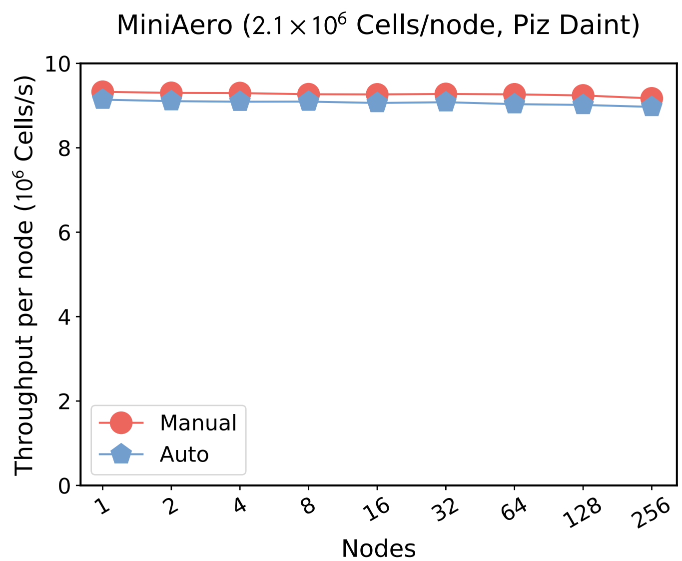
Evaluation

- **Implemented the constraint inference and solver algorithms in Regent**[†]
    - Regent is a high-level programming language with **first-class data partitions and DPL**
- **Weak scaling performance of four benchmark programs**
    - Stencil: 9-point stencil in 2D grid
    - MiniAero: explicit Navier-stokes solver on hexahedral 3D mesh
    - Circuit: circuit simulator on unstructured circuit graphs
    - PENNANT: Lagrangian hydrodynamics on unstructured 2D mesh
- **Machine: Piz Daint** (12-core Xeon E5-2690, NVIDIA P100, and 64 GB memory per node)
- **All benchmark programs ran on GPUs**

† Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, Alex Aiken, *Regent: a high-productivity programming language for HPC with logical regions.* SC15.

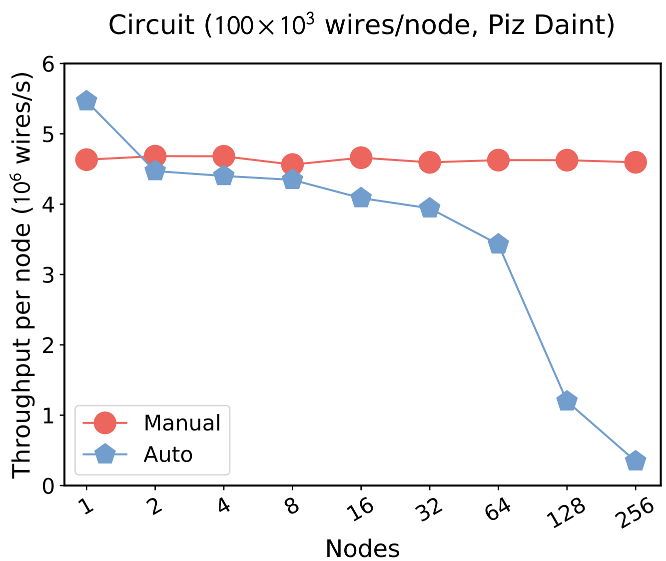
Weak Scaling: Stencil and MiniAero



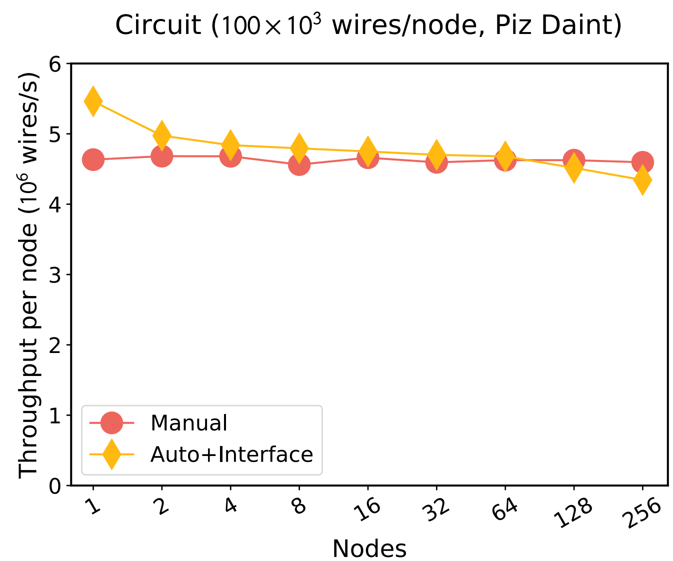


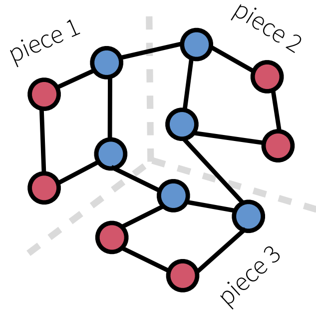
Auto-parallelized programs match hand-parallelized programs within 3%

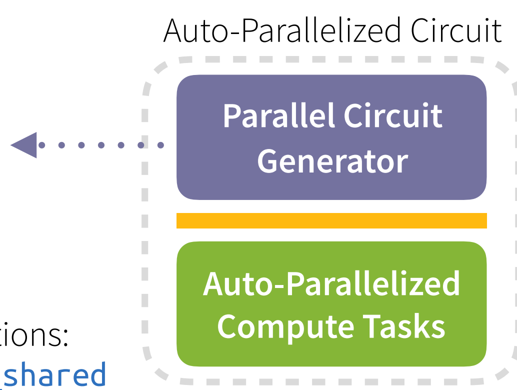
Weak Scaling: Circuit



Add interface constraints





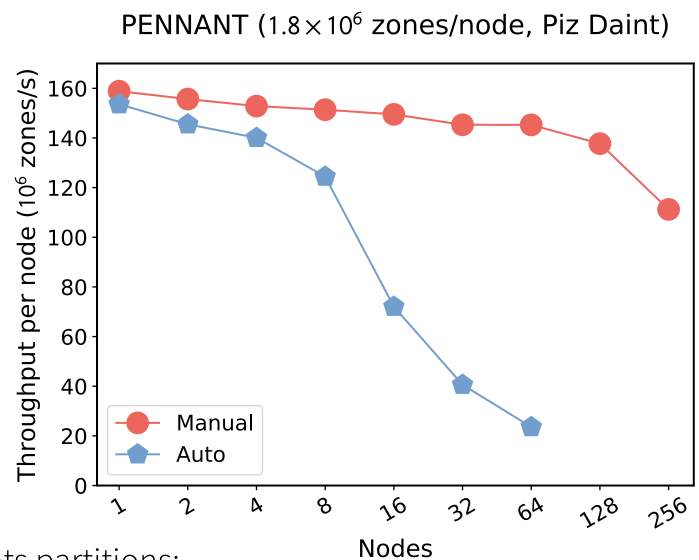


Generator uses two node partitions:
pNodes_private and pNodes_shared

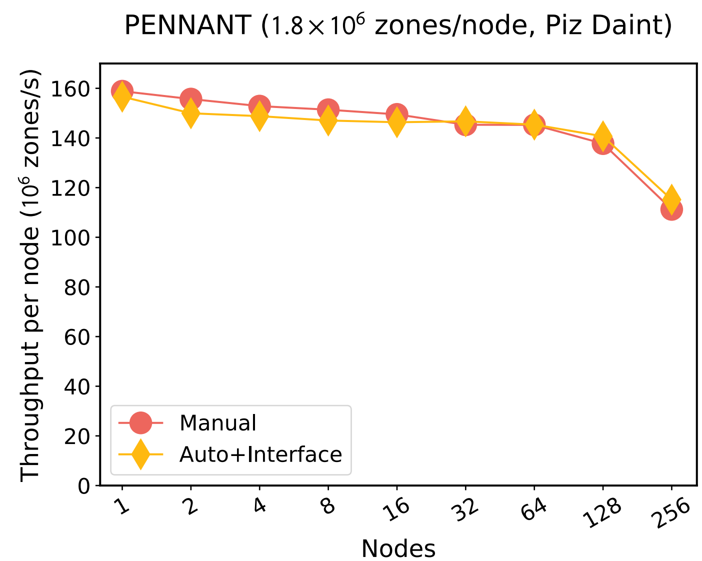
Interface constraints:

`complete(pNodes_private ∪ pNodes_shared,Nodes)`

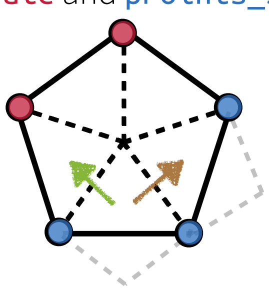
Weak Scaling: PENNANT



Add interface constraints

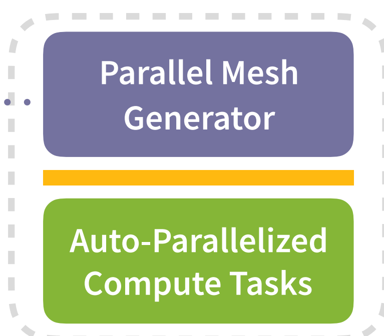


Two points partitions:
`pPoints_private` and `pPoints_shared`



Each side of a polygon colocates with its previous and next sides

Auto-Parallelized PENNANT



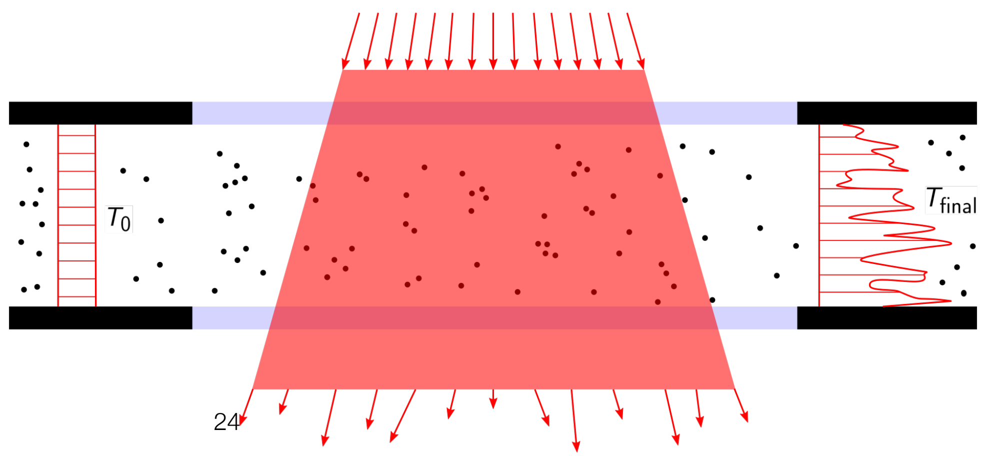
Interface constraints:

```
complete(pPoints_private ∪ pPoints_shared,Points)
image(pSides,Sides[·].prev_side) ⊆ pSides
image(pSides,Sides[·].next_side) ⊆ pSides
```

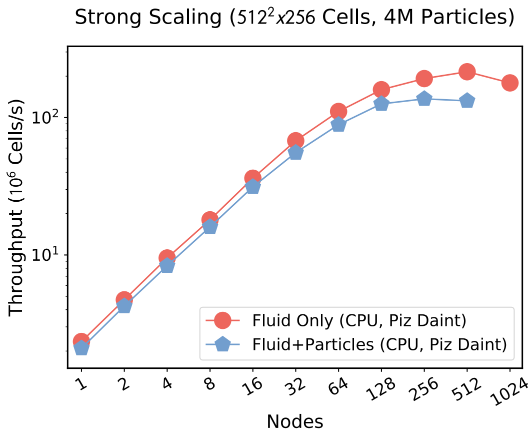
Case Study: Soleil-X

- Developed for the PSAAP II program at Stanford
- Eulerian Fluid + Lagrangian Particles + Radiation (DOM/Algebraic)
  - DOM is manually parallelized
  - Fluid and particles are auto-parallelized except for particle transfers

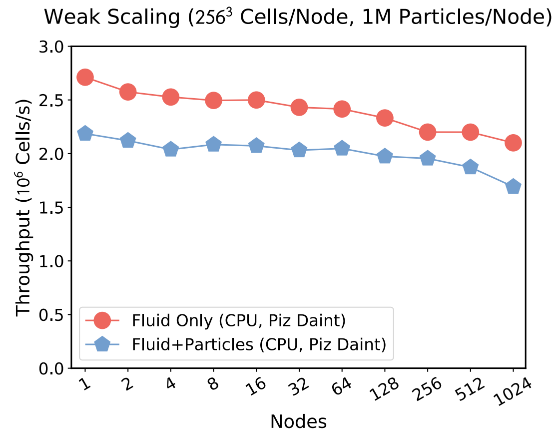
Heated section of concentrated solar energy receiver



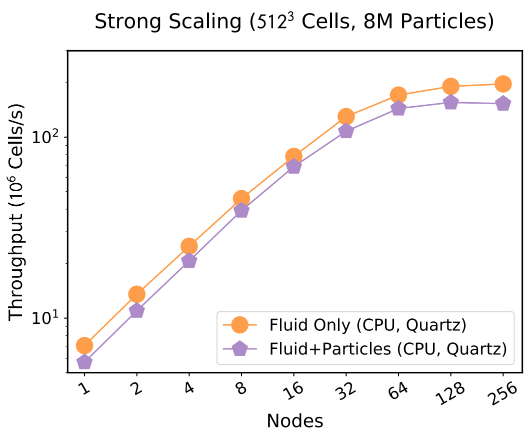
Soleil-X Performance



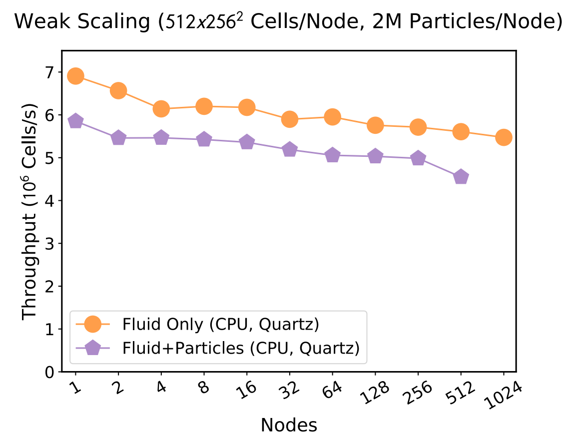
92X speedup at 512 for fluid
65X speedup at 256 with particles



77% parallel efficiency
(9,216 CPUs)

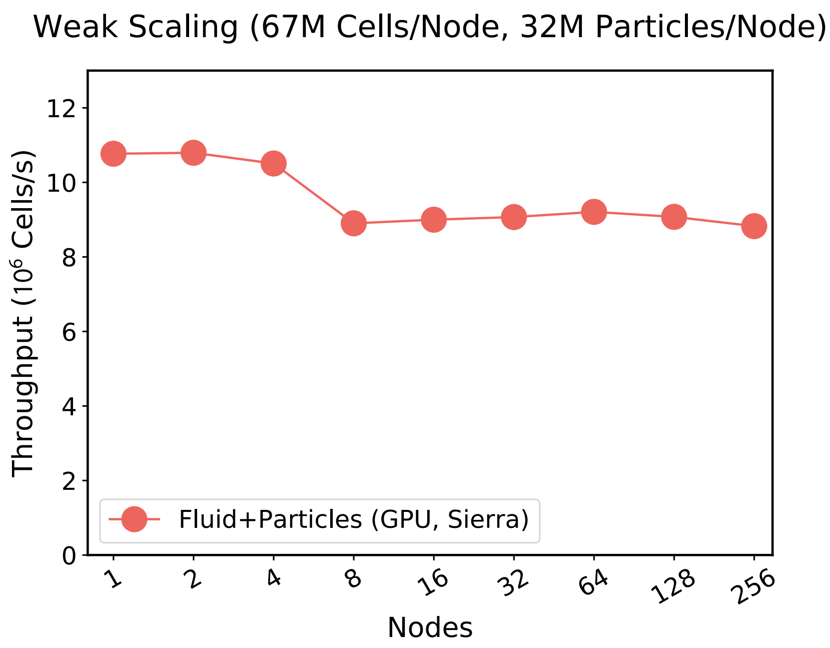


17X speedup at 128 for fluid
15X speedup at 128 with particles



79% parallel efficiency
(30,720 CPUs)

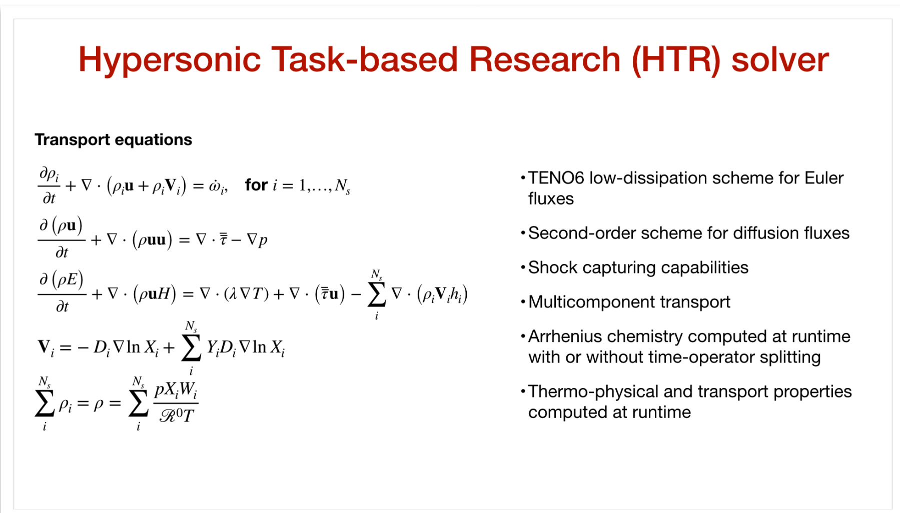
Soleil-X Performance



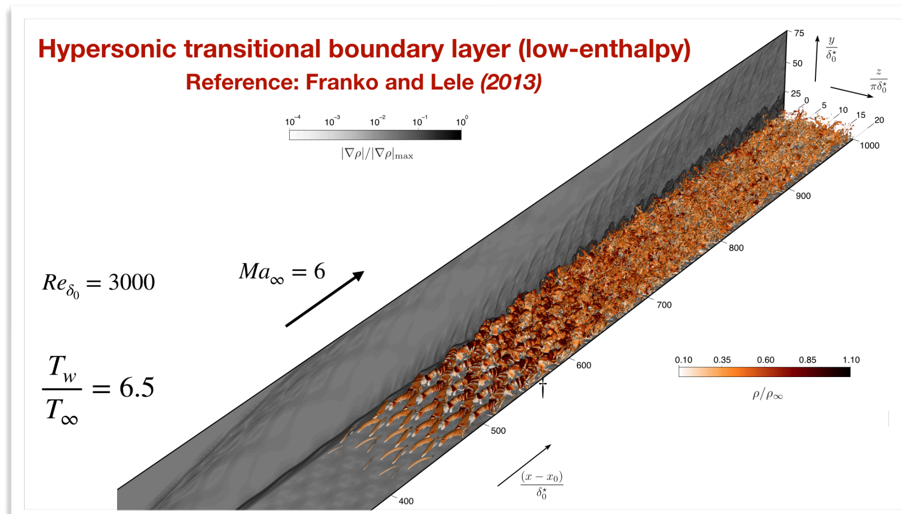
82% parallel efficiency
(1,024 GPUs)

Case Study: HTR Solver

- Solves multi-component Navier-Stokes equations in compressible formulation
  - Accounts for complex chemistry and multicomponent transport
- Heavy flux tasks are auto-parallelized



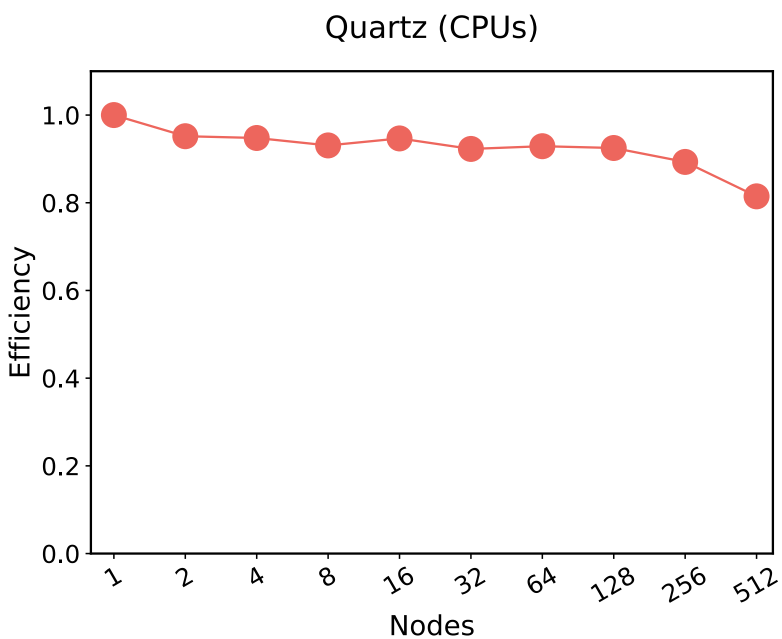
†



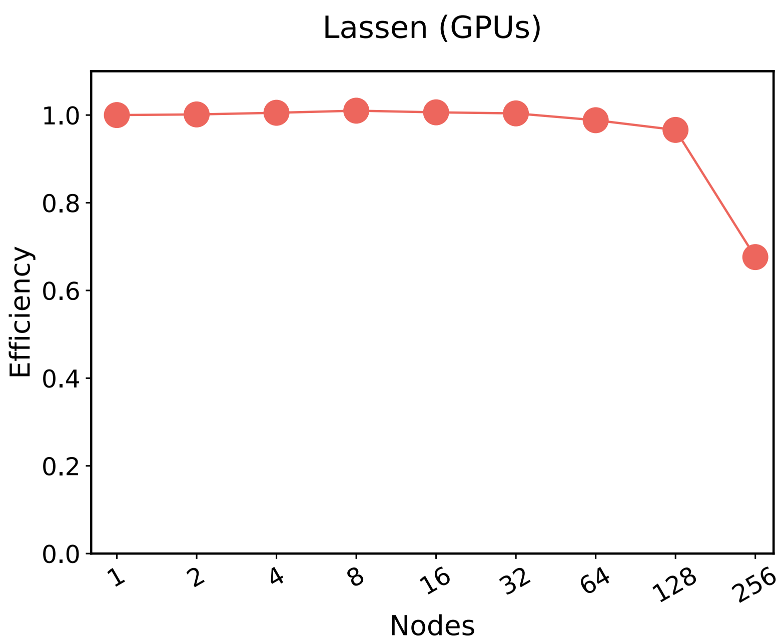
†

† Courtesy of Dr. Mario Di Renzo at Stanford (`mariodr@stanford.edu`)

HTR Performance



81% efficiency
18,432 CPUs
400M points
1s/timestep

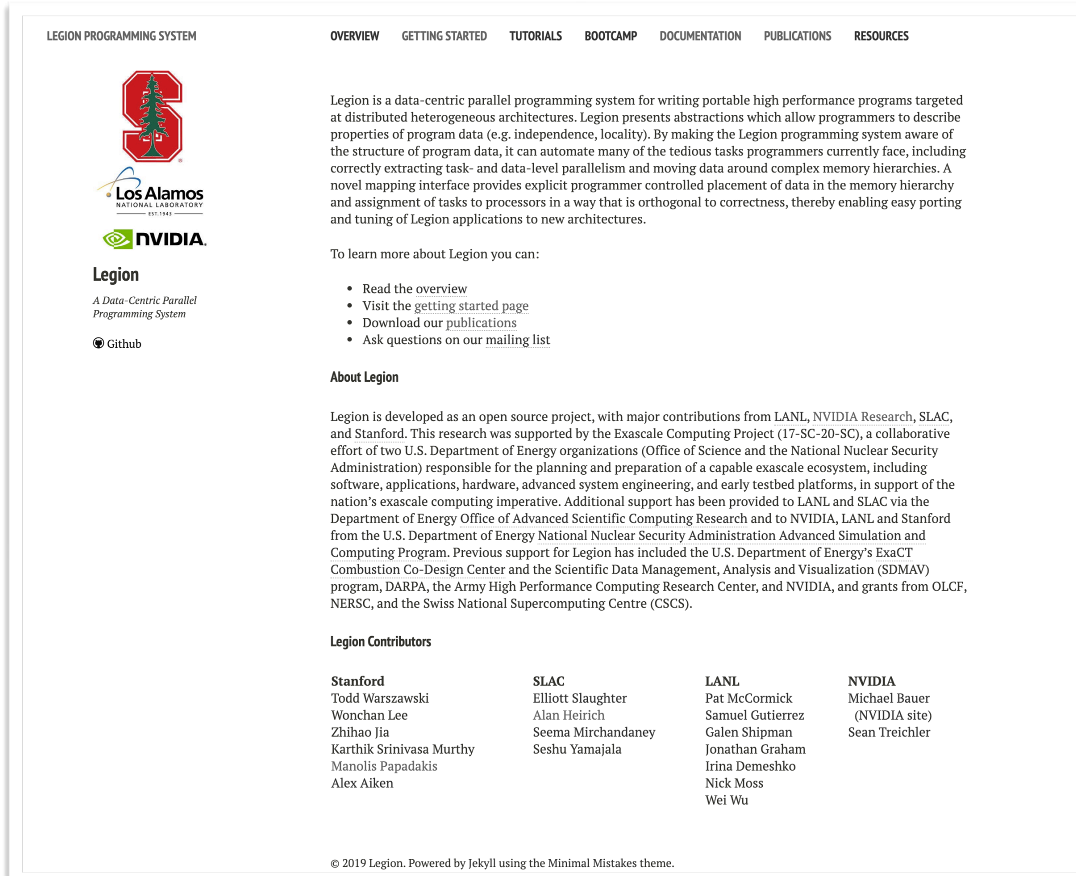


68% efficiency
1,024 GPUs
4.8B points
0.7s/timestep

Conclusion

- First-class data partitions enable composable and configurable auto-parallelization
- A constraint-based data partitioning brings scalability of manual parallelization to auto-parallelized programs

Questions?



LEGION PROGRAMMING SYSTEM

OVERVIEW GETTING STARTED TUTORIALS BOOTCAMP DOCUMENTATION PUBLICATIONS RESOURCES

Legion

A Data-Centric Parallel Programming System

Github

Legion is a data-centric parallel programming system for writing portable high performance programs targeted at distributed heterogeneous architectures. Legion presents abstractions which allow programmers to describe properties of program data (e.g. independence, locality). By making the Legion programming system aware of the structure of program data, it can automate many of the tedious tasks programmers currently face, including correctly extracting task- and data-level parallelism and moving data around complex memory hierarchies. A novel mapping interface provides explicit programmer controlled placement of data in the memory hierarchy and assignment of tasks to processors in a way that is orthogonal to correctness, thereby enabling easy porting and tuning of Legion applications to new architectures.

To learn more about Legion you can:

- Read the overview
- Visit the getting started page
- Download our publications
- Ask questions on our mailing list

About Legion

Legion is developed as an open source project, with major contributions from LANL, NVIDIA Research, SLAC, and Stanford. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative. Additional support has been provided to LANL and SLAC via the Department of Energy Office of Advanced Scientific Computing Research and to NVIDIA, LANL and Stanford from the U.S. Department of Energy National Nuclear Security Administration Advanced Simulation and Computing Program. Previous support for Legion has included the U.S. Department of Energy's ExaCT Combustion Co-Design Center and the Scientific Data Management, Analysis and Visualization (SDMAV) program, DARPA, the Army High Performance Computing Research Center, and NVIDIA, and grants from OLCF, NERSC, and the Swiss National Supercomputing Centre (CSCS).

Legion Contributors

Stanford	SLAC	LANL	NVIDIA
Todd Warszawski	Elliott Slaughter	Pat McCormick	Michael Bauer (NVIDIA site)
Wonchan Lee	Alan Heirich	Samuel Gutierrez	Sean Treichler
Zhihao Jia	Seema Mirchandaney	Galen Shipman	
Karthik Srinivasa Murthy	Seshu Yamajala	Jonathan Graham	
Manolis Papadakis		Irina Demeshko	
Alex Aiken		Nick Moss	
		Wei Wu	

https://legion.stanford.edu